A Principled Approach to REPL Interpreters

L. Thomas van Binsbergen
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands
ltvanbinsbergen@acm.org

Mauricio Verano Merino
Eindhoven University of Technology
Eindhoven, The Netherlands
m.verano.merino@tue.nl

Pierre Jeanjean
Inria, University of Rennes, CRNS, IRISA
Rennes, France
pierre.jeanjean@inria.fr

Tijs van der Storm
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands
University of Groningen
Groningen, The Netherlands
storm@cwi.nl

Benoit Combemale
University of Rennes, Inria, CNRS, IRISA
Rennes, France
benoit.combemale@irit.fr

Olivier Barais
University of Rennes, Inria, CNRS, IRISA
Rennes, France
olivier.barais@irisa.fr

Abstract

Read-eval-print-loops (REPLs) allow programmers to test out snippets of code, explore APIs, or even incrementally construct code, and get immediate feedback on their actions. However, even though many languages provide a REPL, the relation between the language as is and what is accepted at the REPL prompt is not always well-defined. Furthermore, implementing a REPL for new languages, such as DSLs, may incur significant language engineering cost.

In this paper we survey the domain of REPLs and investigate the (formal) principles underlying REPLs. We identify and define the class of sequential languages, which admit a sound REPL implementation based on a definitional interpreter, and present design guidelines for extending existing language implementations to support REPL-style interfaces (including computational notebooks). The obtained REPLs can then be generically turned into an exploring interpreter, to allow exploration of the user's interaction.

The approach is illustrated using three case studies, based on MiniJava, QL (a DSL for questionnaires), and eFLINT (a DSL for normative rules). We expect sequential languages, and the consequent design principles, to be stepping stones towards a better understanding of the essence of REPLs.

CCS Concepts: • **Software and its engineering** → **Interpreters**; *Domain specific languages*; Integrated and visual development environments; • **Human-centered computing** → *Command line interfaces.*

Onward! '20, November 18–20, 2020, Virtual, USA

ACM ISBN 978-1-4503-8178-9/20/11…$15.00
https://doi.org/10.1145/3426428.3426917

Keywords: interpreters, REPLs, software language engineering, notebooks, meta-languages, language workbenches

ACM Reference Format:
L. Thomas van Binsbergen, Mauricio Verano Merino, Pierre Jeanjean, Tijs van der Storm, Benoit Combemale, and Olivier Barais. 2020. A Principled Approach to REPL Interpreters. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '20), November 18–20, 2020, Virtual, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3426428.3426917

1 Introduction

"The top level is hopeless", Matthew Flatt[1]

Read-eval-print-loops (REPLs, also known as command-line interfaces, or interactive shells) are a popular way for programmers to interact with programming languages. They allow incremental definition of abstractions, testing out snippets of code with immediate feedback, debugging executions, and explore APIs.

Some languages, such as scripting languages or interpreted languages, are more naturally compatible with the REPL mode of interaction and the styles of programming that it enables (and that programmers have come to expect). For example, a sequence of valid code snippets written in the REPL of Python is itself a valid Python program. On the other hand, JShell, for instance, allows programmers to write expressions, statements, variable declarations and method declarations as code snippets, even though these constructs are not allowed at the top-level in Java programs.

Consider the following example JShell interaction (every line is a code snippet sent separately):

```
class Example {}
Example obj = new Example();
class Example { public int meth() { return var; } }
int var = 1;
```

[1] https://gist.github.com/samth/3083053

This example raises the questions whether classes can be redefined, whether `obj` can be accessed after `Example` is redefined or if `obj` is migrated, and, if so, what methods it has and, if `meth` is available, whether a call `obj.meth()` returns `1`. Without giving answers here, the example shows that the relation between a programming language and the behavior of its REPL is not immediately obvious. Matthew Flatt's *ceterum censeo* quoted above bears witness to the fact that the relation can actually be strenuous and cause a lot of confusion. The above questions are fundamentally about language design: several sensible answers are possible and the answers have a significant impact on programmer experience.

In some sense, JShell can be seen to implement *its own* language, which, even though strongly reminiscent of Java, is markedly different. In this paper, we take this observation and run with it: we assume that a REPL interpreter for $\mathcal{L}$ effectively defines its own language $\mathcal{R}$, often as an extension or modification of $\mathcal{L}$, whose programs are sequences of valid code snippets according to the REPL.

To this end we identify and define the class of languages that underlie REPL interpreters as *sequential languages*. The essence of sequential languages is that the *concatenation of two programs is again a program*. Or, to put it more precisely, a language is sequential if it features an associative sequencing operator $\fatsemi$, such that the following equation holds:

$$[\![p_1 \fatsemi p_2]\!] = [\![p_2]\!] \circ [\![p_1]\!]$$

The meaning of a sequence of program fragments is defined by composing the meanings of the individual fragments, including any impure effects of these fragments.

The notion of sequential language informs a methodology to make a language sequential, and hence suitable for sound REPL interpreters. The methodology enforces certain design principles on the REPL engineer to ensure that questions like the ones asked about the JShell interaction are answered precisely and are explicitly addressed as matters of language design, instead of an implementation concern. Furthermore, sequential languages are amenable to interfaces which allow *exploring* execution traces resulting from REPL interactions.

We have applied this methodology in three case studies. The first extends an existing implementation of MiniJava [1] in the Rascal language workbench [15], to make it sequential. This extended MiniJava is then the base interpreter for a computational notebook interface through Bacatá, Rascal's bridge to Jupyter [22]. The second case study involves QL, a DSL for defining spreadsheet-like interactive questionnaires [8, 9]. This case study show that it is feasible to obtain REPLs for languages that are not statement- or expression-oriented. The third case-study applies the methodology to obtain interactive services for eFLINT, a DSL for executable normative specifications [43]. The resulting services allow users and policy-aware software to navigate choices and decisions in the realm of law and regulation.

To summarize, the contributions of this paper are:

Table 1. Surveyed REPL implementations.

REPL	Reference
CLing (C/C++)	https://cdn.rawgit.com/root-project/cling/master/www/index.html
JShell (Java)	http://openjdk.java.net/jeps/222
Python	https://docs.python.org/3/tutorial/interpreter.html
C#	https://www.mono-project.com/docs/tools+libraries/tools/repl/
Node.js (Javascript)	https://nodejs.org/api/repl.html
PHP	https://www.php.net/manual/en/features.commandline.interactive.php
PsySH (PHP)	https://psysh.org/
SQLite (SQL)	https://sqlite.org/
R	https://www.r-project.org/
Swift	https://swift.org/lldb/
Gore (Go)	https://github.com/motemen/gore
GNU Octave	https://www.gnu.org/software/octave/
Rappel (assembly)	https://github.com/yrp604/rappel
iRB (Ruby)	https://github.com/ruby/irb

- A feature-based analysis of the landscape of REPLs for a selection of the most popular programming languages (Section 2).
- A formalization of the notion of sequential language as the underlying principle of REPLs (Section 3)
- A language-parametric *exploring interpreter* algorithm on top of existing interpreters, allowing users to navigate user interaction history (Section 4)
- A methodology for developing REPL interpreters by *sequentializing* languages with a definitional interpreter (Section 5).
- Three case studies to illustrate the feasibility of the approach (Section 6).

The paper is concluded with a discussion of limitations, related work, and directions for further research.

2 REPL Domain Analysis

This section provides a study of existing REPL interpreters and their main features. We have studied freely available REPL implementations, listed in Table 1, for the 15 most popular languages from the TIOBE index[2], with the exception of Visual Basic, for which we could not find an freely available implementation. For MATLAB we have selected GNU Octave as a substitute. We performed a feature-oriented domain analysis [14], resulting in the feature model of Figure 1. Below we briefly describe the main mandatory and optional features.

Mandatory Features. An interpreter must have certain features to be considered a REPL. In particular, a REPL has the ability to execute multiple code snippets across multiple

[2] https://www.tiobe.com/tiobe-index/ (accessed May, 22nd, 2020)

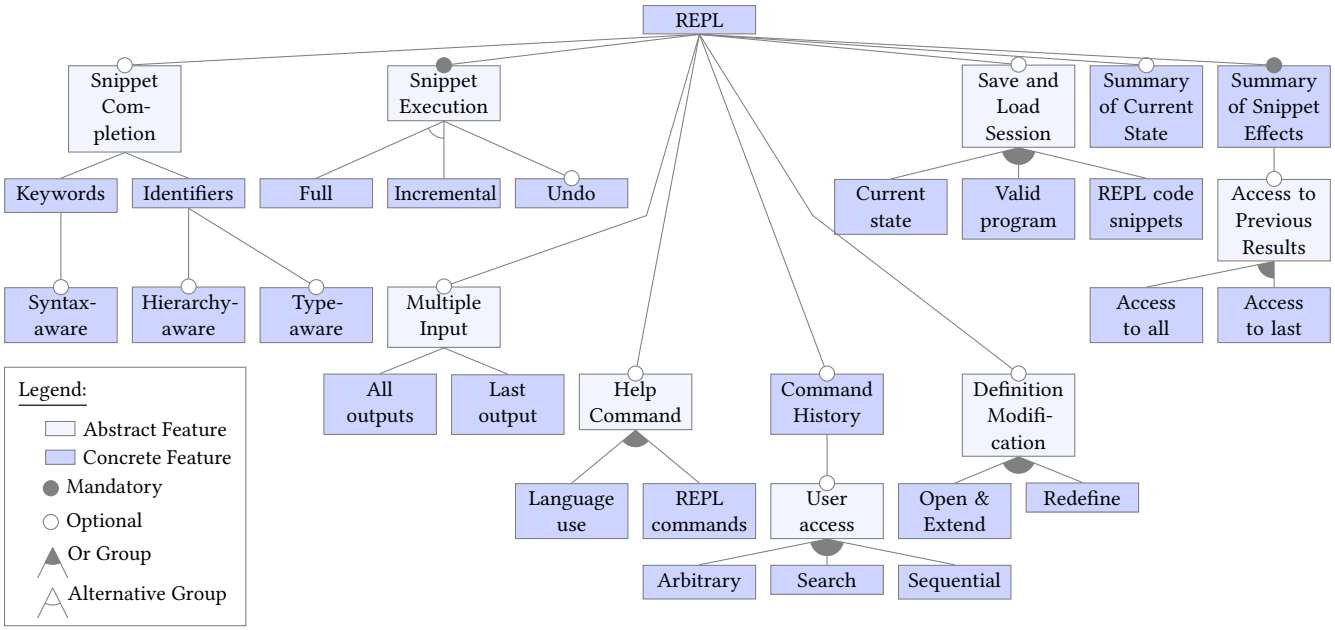

Figure 1. Feature Model for REPL Interpreters

interactions in a single session (as opposed to executing one full program per session). In most of the investigated REPL implementations, the REPL maintains execution context and executes snippets incrementally (the "Incremental" alternative of the "Snippet Execution" feature). Optionally, a REPL may provide a way to undo the execution of snippets (rollback). An alternative to incremental execution is composing all snippets into a single program and execute the program from scratch (the "Full" alternative). REPLs are expected to provide feedback after evaluating snippets, showing at least the snippet's printed output, and perhaps any result values or newly declared types ("Summary of Snippet Effects").

Optional Features. Next to these mandatory features, the investigated REPLs implement several additional features such as auto-completion of snippets ("Snippet Completion"). This can target either language keywords or previously defined identifiers. Completion can take into account the syntactic context in which the user is typing, can be extended to fully qualified identifiers, and may also take into account the type of identifiers (through static typing or type hinting).

Even though the language itself might not support modifying an existing definition ("Definition Modification"), most REPLs allow this behavior to some extent. Common ways include overriding the previous definition, either through a new definition snippet or by editing it from an external text editor. Other REPLs also allow opening up definitions (such as classes) for additions ("Open & Extend").

Another common feature is the help (meta-)command ("Help Command"), which can document either the language, the REPL and its meta-commands, or both. The history of commands (including snippets) is usually made available to the user, in order to find and resubmit previous commands ("Command History"). It can be consulted sequentially through the arrow keys, but often includes a search facility as well. Some REPLs assign identifiers to commands in order to retrieve them arbitrarily. Some REPLs support saving and loading sessions ("Save and Load Session"). This may involve storing the execution context, or simply storing all user inputs to reproduce the execution context after loading. For some languages, the session can also be saved as a valid program outside of the REPL.

REPLs behave differently when multiple code snippets are input at once ("Multiple Input"). Output is either provided for all of the snippets or only for the last snippet (which could result in no output at all). Most REPLs allow inspection of the current execution context to the user ("Summary of Current State"). And finally, some REPLs allow the results of previous snippets to be used in new snippets ("Access to Previous Results"), either for the last executed snippet or for all by, for instance, assigning result values to variables.

Feature Support of Existing REPLs. Table 2 shows how the investigated REPLs support the features identified in the feature model of Figure 1. The table illustrates that no two REPLs share the same set of features. IPython supports most features, whereas PHP supports a minimal set of features. Interestingly, PHP is the only REPL that does not print computed output values. The Go REPL (Gore) is the only REPL

Table 2. REPL Interpreter Features (●= full, ◒ = partial, – = not applicable)

		CLing	JShell	Python	IPython	C# REPL	Node.js	PHP	PsySH	SQLite	R	Swift	Gore	Octave	Rappel	iRB
Snippet Execution	Incremental	●	●	●	●	●	●	●	●	●	●	●		●	●	●
	Full												●			
	Undo	●														
Summary of Current State		●	●		●	●			●	●				●	●	
Summary of Snippet Effects		●	●	●	●	●	●		●	●	●	●	●	●	●	●
Access to Previous Results	Access to last			●	●		●		●					●		●
	Access to all		●		●							●				
Multiple Input	Last output	●			●	●	●		●			●		●	●	●
	All outputs		●	●						●	●					
Snippet Completion	Keywords	●		●	●		●			●	●	●		●		●
	Syntax-aware	●			●							●				
	Identifiers	●	●	●	●	●	●	●	●	●	●	●	●	●		●
	Type-aware		●				–				–				–	
	Hierarchy-aware	●	●	●	●	●	●		●		●		●	●	–	
Definition Modification	Redefine		●[1]	●	●[1]	●					●[1]	●	●	●[1]	–	●
	Open & Extend														–	●
Help Command	REPL commands	●	●	●	●	●	●		●	●		●	●		●	
	Language use			●	●						●		●	●		
Command History (User Access)	Sequential	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
	Search	●	●	●	●	●	●	●	●	●	●		●	●		●
	Arbitrary		●		●									●		
Save and Load Session	Current state									●	●			◒		
	REPL code snippets		●		●		●			◒	◒					
	Valid programs				●		●			◒	◒		●			

[1] The previous definition can be opened in an external editor for editing

that simulates incremental execution by compiling a complete compilation unit in the back-ground. Type-aware completion is not applicable to Node.js and R since the languages are dynamically typed and do not support type hinting. Sessions exported from SQLite and R include the snippets to reproduce data, but not the results of querying the data. Octave exports variables and their values, but not declared methods. Only three REPLs support exporting sessions as valid programs. Although IPython provides additional commands, they are all implemented in Python and can therefore be exported. As explained before, a valid Go program is produced as part of every interaction with Gore. The interactive interpreter for Swift also provides debugging facilities. This feature was observed but not discussed as a REPL feature because the behaviors are accessed by running the interpreter in different 'modes'. Interestingly, the decision to provide both modes in a single tool was made from observing that the modes shared similar features such as expression evaluation, data monitoring and step by step execution.

The wealth of features and diversity observed in REPLs motivated this paper's study into the foundations of REPLs.

3 Sequential Languages

This section defines the class of software languages for which the semantics can be expressed as a deterministic transition relation (a transition function). A subclass of these languages – the so-called *sequential languages* – is defined as the set of languages in which programs are written as sequences of smaller programs. A language is defined as a set of syntactically valid programs[3] with an interpreter assigning to each program the *effect* of the program, expressed as mutations on the context in which it is executed. The context is called a *configuration* in reference to Plotkin's Structural Operational Semantics [30]. A program's effect is thus modeled as a function from configuration to configuration[4]. This model is sufficient to describe the semantics of real-world, large-scale, deterministic programming languages as is demonstrated by the body of literature on big-step, small-step and natural semantics [2, 13, 24, 26, 30] and does not exclude languages with non-deterministic aspects when these aspects can be captured algebraically [46].

Definition 3.1. A language L is a structure $\langle P, \Gamma, \gamma^0, I \rangle$ with P a set of programs, Γ a set of configurations, $\gamma^0 \in \Gamma$ an initial configuration and I a definitional interpreter assigning to each program $p \in P$ a function $I_p : \Gamma \to \Gamma$.

[3] The abstract syntax of the language.

[4] Corresponding to a big-step transition relation or the transitive closure of a small-step transition relation.

Definition 3.2. A language $L = \langle P, \Gamma, \gamma^0, I \rangle$ is *sequential* if there is an operator ; such that for every $p_1, p_2 \in P$ and $\gamma \in \Gamma$ it holds that $p_1; p_2 \in P$ and that $I_{p_1;p_2}(\gamma) = (I_{p_2} \circ I_{p_1})(\gamma)$.

Any two programs of a sequential language can be combined to form a new program whose effects are equal to the composition of the effects of the individual programs. If a language L does not have an operator ; with $I_{p_1;p_2} = I_{p_2} \circ I_{p_1}$, then the language is easily extended to have such an operator by taking $I_{p_1;p_2} = I_{p_2} \circ I_{p_1}$ as the definition of its semantics.

If the set P of the definition of a sequential language L is taken as the set of code snippets accepted by the REPL for L, then the ; operator describes the (snippet-related) behavior of the REPL. This is under the assumption that REPLs should always accumulate the effects of the code snippets they are asked to execute. The definitional interpreter of the language determines the effects of individual code snippets as well as the effect of their compositions. These observations show that, when it comes to the syntax and semantics of code snippets, REPL engineering can be considered as a matter of language design and engineering. On top of this there are several benefits to basing a REPL on a sequential language, i.e. to having the sequential composition operator as a language construct. These benefits are discussed throughout this paper.

The ; operator of the Definition 3.2 does not necessarily correspond to the sequence operator of imperative or statement-based languages (often written as a semicolon). This is best exemplified by errors and exceptions. A sequence of statements typically terminates upon the occurrence of an exception, whereas a REPL is not expected to terminate if a code snippet raising an exception is submitted. As an example, consider the following JShell interaction:

```
jshell> (1/0); System.out.println(5)
|  Exception java.lang.ArithmeticException: / by zero
|        at (#1:1)
jshell> (1/0)
|  Exception java.lang.ArithmeticException: / by zero
|        at (#2:1)
jshell> System.out.println(5)
5
```

This example shows that the effect of executing a snippet throwing an exception is to output information about the exception and to ensure that control flow returns to normal. In the first snippet, the print-statement is not executed, because of the exception. But after the second snippet, the user can continue printing a value.

Definition 3.3. Given a language $L = \langle P, \Gamma, \gamma^0, I \rangle$, the *reachability graph* from $\gamma \in \Gamma$ is the graph $\langle V, E \rangle$ with V and E the smallest sets of nodes and labeled edges such that $\gamma \in V$ and for every triple $\langle \gamma_1, p, \gamma_2 \rangle$, with $\gamma_1 \in V$ and $\gamma_2 = I_p(\gamma_1)$, it holds that $\gamma_2 \in V$ and that $\langle \gamma_1, p, \gamma_2 \rangle \in E$.

The reachability graph encodes, as paths, every possible execution run resulting from executing some sequence of programs in the context of configuration γ.

Lemma 3.4. *The reachability graph from any configuration in a sequential language is closed under transitivity, with $\langle \gamma_1, p_1; p_2, \gamma_3 \rangle \in E$ if and only if there exists a $\gamma_2 \in \Gamma$ with $\langle \gamma_1, p_1, \gamma_2 \rangle \in E$ and $\langle \gamma_2, p_2, \gamma_3 \rangle \in E$.*

Proof. Follows from the definitions of sequential languages and reachability graphs. □

The effect of a program can be defined as the difference between the source and target configuration of an edge in a reachability graph, i.e. if $\langle \gamma, p, \gamma' \rangle$ is in some reachability graph, the effect of p in γ is the difference between γ' and γ. Lemma 3.4 states that the reachability graph for a sequential language is closed under transitivity. It follows that the effects of a *path* in the reachability graph, i.e. the effects of the sequence of programs occurring as labels on the edges of that path, is simply the difference between the start and end configuration of the path. The effects of a path can thus be computed without the need to compute the effects of all individual programs separately. Moreover, every path describes a valid program that can be saved (possibly together with its effects). Sequential languages thus admit simple implementations of the "Save and Load Session" feature of REPLs.

4 Exploring Interpreters

This section defines a generic algorithm for executing programs by calling an underlying definitional interpreter and recording the resulting configurations. Specialized to a particular language $\mathcal{L}$, this algorithm is the *exploring interpreter* for $\mathcal{L}$. The exploring interpreter for $\mathcal{L}$ records configurations in a subgraph of the reachability graph for $\mathcal{L}$ and is capable of reverting to a recorded configuration. Exploring interpreters admit exploratory programming by enabling programmers to revert to previous execution states in order to explore the effects of alternative sequences of code snippets.

The formal definition of exploring interpreters follows naturally from the definitional interpreter component of the definition of languages. However, definitional interpreters that form the basis for exploring interpreters have much stronger implementation requirements than definitional interpreters for REPLs without exploration. In particular, exploring interpreters require all program effects to be represented by changes in data, i.e. the definitional interpreter has to be a pure function. Without exploration, REPL implementations can be based on definitional interpreters that use real, rather than simulated, IO, memory and network communication (while still respecting their formal definition). Although exploring interpreters form the basis of the MiniJava and eFLINT case studies in this paper, the core of the methodology we propose in Section 5 is also applicable

to implementations without explicit state representation (as discussed in Section ??).

Definition 4.1. An *exploring interpreter* for a language $\langle P, \Gamma, \gamma^0, I \rangle$ is an algorithm maintaining a current configuration (initially γ^0) and an *execution graph* (initially containing just the node γ^0) and iteratively executing one of the following actions. At any moment the execution graph is a subgraph of the reachability graph from γ^0.

- **execute**(p): transition from the current configuration γ to the configuration $\gamma' = I_p(\gamma)$, where $p \in P$ is provided as input, and subsequently:
  - add γ' to the set of nodes (if new),
  - add $\langle \gamma, p, \gamma' \rangle$ to the set of edges (if new),
- **revert**(γ): take γ as the current configuration for the next action, where $\gamma \in \Gamma$ is provided as input
- **display**: produce a structured representation of the current graph, distinguishing the current configuration in the graph from the other configurations.

The exploring interpreter algorithm is generic in that it has the language components P, Γ, I and γ^0 as inputs (type parameters in implementations). The **display** action can be used by interfaces to visualize the execution graph and to enable users to choose a node to **revert** to (see the MiniJava notebook discussed in Section 6).

REPL interpreters typically do not support the kind of exploration enabled by the **revert** action (only CLing has the "Undo" feature of the feature model). For practical purposes, such as space-efficiency, it may be desirable to implement a less powerful version of the algorithm in which the execution graph is maintained as a tree, a single path or even just a single node. For example, an implementation of **revert**(γ) that removes all descendants of γ requires less space as it maintains only a single path (the exploring interpreter behaves like a stack). For another example, if a **execute**(p) is implemented to always create a new node for every γ', then the execution graph is actually a tree in which multiple nodes may hold the same configuration. There are advantages to both execution graphs (with sharing) and execution trees (without sharing). With sharing, there is the potential to avoid redoing (potentially costly) computations. This situation arises if the current node with configuration γ has an outgoing edge labeled q. If in this situation the action **execute**(q) is performed, then there is no need to interpret q as there is already an edge $\langle \gamma, q, \gamma' \rangle$ in the graph (for some γ'). The potential to avoid costly computations significantly increases if the graph is kept closed under transitivity (which is possibly for sequential languages according to Lemma 3.4) and if program transformations are used to label the edges with normal forms. Without sharing, there is exactly one path from the root node to every other node, i.e. every node has a unique 'history'. By reverting to a particular node, the user has not only chosen a configuration, but also the sequence of programs that led to that configuration. This is helpful, for example, when printing the effects of the snippets that gave rise to the current configuration after a **revert**. If, after reverting, the current node has multiple incoming edges, then it is not clear what output should be printed.

5 Methodology

In this section we propose a methodology for developing REPL interpreters based on the definitions and observations of the previous sections. The methodology proposes to build a REPL for some base language on top of an exploring interpreter for a sequential language defined as an extension of, or modification to, the base language. An exploring interpreter is essentially a bookkeeping device on top of a definitional interpreter and provides the "Incremental Snippet Execution" and "Undo" features directly (cf. Section 2). Additional motivation for using the exploring interpreter (for a sequential language) is that it promotes certain design principles while preserving the ability to implement many desirable features. These principles and their consequences are discussed in this section, together with a summary of the proposed methodology.

The core principles underlying our methodology are:

- the effects of a code snippet manifest as changes to an *explicit* state representation (a configuration)
- the effects of a code snippet are determined by the definitional interpreter used by the exploring interpreter
- the effects of a sequence of code snippets is the composition of the effects of the individual snippets
- only code snippets change configurations

For users of the REPL, the most important consequence of these principles it that an understanding of the definitional interpreter is enough to understand the precise behavior of the REPL for the language. In practical terms: to know the effects of code snippets, a user needs to understand the base language and its extension or modification to a sequential variant. The extension or modification is made explicit by the definitional interpreter and should be communicated clearly (as precise documentation, a formal semantics, or an open-source implementation).

For engineers of the REPL, the most important consequence of the principles is that every feature (on top of "Incremental Snippet Execution" and "Undo") is implemented either:

- as a language extension (e.g. the features "Definition Modification" and "Access to Previous Results"),
- as a series of interactions with the exploring interpreter (e.g. "Multiple Input", explained below),
- based on information stored in the execution graph (e.g. "Summary of Snippet Effects", "Summary of Current State" and "Snippet Completion") or
- independently of the exploring interpreter, when the feature does not involve snippet execution (e.g. "Help Command" and other meta-commands).

The methodology of this paper is based on the hypothesis that many of the features of existing REPLs, including at least those in Figure 1, fall into the four categories listed above. This hypothesis is tentatively supported by the various feature implementations described across Section 6.

To prelude the example feature implementations of Section 6, consider the alternatives of the "Multiple Input" feature ("All outputs" and "Last output"). A "Multiple Input" snippet is parsed as, for example, $p; q; r$. In the implementation of a REPL following our methodology, such a snippet can be handled by performing three **execute** actions with respectively p, q and r as inputs (because the language is sequential). The REPL has then seen the four configurations γ_0, γ_p, γ_q and γ_r corresponding to the configuration before executing p, after executing p, after q and after r respectively. The output of the last input r is found by computing the difference between γ_q and γ_r, the output of all three inputs is found by computing the difference between γ_0 and γ_r.

The methodology for developing a REPL for any base language $\mathcal{L}$ is formulated as the following steps (and has certain commonalities with the approach of [12]).

1) Definitional Interpreter. Formulate $\mathcal{L}$ as a language in terms of its concrete and abstract syntax, and a definitional interpreter that captures the effects of programs as a function over some set of configurations, thus forming the components of a language according to Definition 3.1. If the language is sequential according to Definition 3.2, then steps 2–5 can be skipped.

2) Phrase Nonterminal. To define a sequential variant $\mathcal{L}'$ of $\mathcal{L}$, reuse the syntax definitions of the previous step to define a new sort **phrase** with an alternate for each of the sorts of $\mathcal{L}$ that describe the syntax of a valid code snippet of the envisioned REPL. The syntax can also have other extensions or modifications, as long as **phrase** is the entry point of the syntax (the first component of a language in Definition 3.1).

3) Phrase Interpreter. Define a definitional interpreter for $\mathcal{L}'$ to capture the semantics of phrases, reusing as much as possible the definitional interpreter of step 1, ideally by applying modular extension mechanisms (e.g., Object Algebras [11, 27], Rascal's `extend` [4]). Special consideration needs to be given to the effects of phrases to ensure the next phrase is executed in the right context. For example, if the result value of a phrase needs to be available to the next phrase through a binding, this binding needs to be introduced as one of the effects of the first phrase.

4) ";"-Phrase. Extend the sort **phrase** with an alternate that combines two valid phrases to form a phrase. For example, with the semicolon as a separator, let $p; q$ be a valid phrase if p and q are valid phrases.

5) Interpreter for ";" Extend the definitional interpreter of $\mathcal{L}'$ such that the effect of a phrase formed by combining two phrases is the composition of the effects of the combined phrases, e.g. $I_{p;q} = I_q \circ I_p$. The language $\mathcal{L}'$ is sequential by definition as a result of this and the previous step.

6) Instantiate Explorer. Obtain an exploring interpreter for $\mathcal{L}'$ by instantiating the generic exploring interpreter algorithm with the definitional interpreter for $\mathcal{L}'$. The implementation may be simplified compared to Definition 4.1 in that it maintains a simpler form of execution graph, if desirable. Instead of an exploring interpreter, the definitional interpreter for $\mathcal{L}'$ can also be used directly. In fact, any implementation that respects the semantics of the definitional interpreter can be used, e.g. an implementation with real rather than simulated effects.

The chosen interpreter can then be offered through various user interfaces, such as command-line interfaces, a network service, or a computational notebook. The interface displays visualizations of the effects of phrases, e.g., by showing output, computed values and new bindings, and can optionally implement additional REPL features.

5.1 Pragmatics

In the context of language workbenches [9] and DSLs [23], a common language implementation strategy is to define interpreters, consisting of functions traversing an abstract syntax tree whilst modifying a propagated configuration to express effects (following the Visitor design pattern). The case studies of the next section include such interpreters. The REPLs in these case studies are obtained through generic implementations of the exploring interpreter algorithm (in Java and in Haskell) that are easily specialized by providing the entry points of the abstract syntax and the interpreter. The presented methodology is based on an exploring interpreter because it is a relatively natural and simple layer to add on top of the described definitional interpreters typically built with Rascal [4, 15]. Moreover, the generic exploring interpreter forms a suitable abstraction for reasoning about sequences of interactions between programmer and REPL – e.g. saving and loading sessions and extracting base language programs – and for implementing advanced REPL and notebook features that support exploratory programming and live programming.

In theory, our approach can also be used for developing REPLs for (general-purpose) programming languages, as many languages can have their semantics expressed as a transition function. In practice, however, very few programming languages have an interpreter implemented as a pure function or have a complete operational semantics from which such an interpreter can be derived. REPLs are not typically implemented with explicit state representation and few enable backtracking (in our survey only CLing supports "Undo"). However, an impure interpreter implementation

can be used at step 6 (Instantiate Explorer) of the methodology. Although some advanced features – such as "Undo" – may then be harder to realize, the most important principles of our methodology still hold. In particular, the differences between base language and REPL should be formulated as extensions or modifications of the base language. This is achieved by updating the semantics of the base language such that repeated execution of its interpreter (i.e. the composition of effects) gives the behavior expected of the REPL of the language. The details of how this can be achieved depend on the language and the techniques used to implement the language. Discussed next are the general patterns that have been observed in our survey.

5.2 Common REPL Language Extensions

As mentioned in Section 3, languages rarely provide an operator that corresponds precisely to the REPL top level. For example, a snippet with an uncaught exception is not expected to prevent subsequent snippets from being executed, whereas termination is expected when an exception occurs within a sequence of (;-separated) statements. Of the surveyed REPLs, only Gore prevents subsequent snippets from executing once a previous snippet raises an exception (a consequence of its "Full" execution model). In the other languages, the REPL top level catches any otherwise uncaught exceptions and presents them to the programmer after which a subsequent snippet can be executed. In languages with constructs for catching and handling exceptions, one might explain or implement this feature with a top-level catch and a handler that prints the exception. For example, a snippet `{System.out.println(1); (1/0);}` can be considered as implicitly wrapped in a **`try`**/**`catch`** block in JShell as follows:

```
try {{System.out.println(1); (1/0);}} catch (Exception e) {
  ... // print the exception in a helpful format
}
```

This clarifies, in reference to the Java semantics, that any effects produced by a snippet before, but not after, an exception is thrown are preserved. However, the translation is inaccurate as a JShell snippet is not an isolated block, unlike a **`try`**-block. Bindings produced by top-level declarations are active when subsequent snippets are executed, i.e. all snippets are in the same scope and the top-level catching exceptions does not change this. In the next JShell fragment, the meta-variable `$1` is available to subsequent snippets despite the exception.

```
jshell> 5; (1/0);
$1 ⇒ 5
|  Exception java.lang.ArithmeticException: / by zero
|        at (#2:1)
```

This example also highlights the importance of presenting new bindings, assignments, and any other effects to the programmer, providing the information required by the programmer to update their mental model of the REPL's execution state.

Another common example of a modification to the base language is the "Access to Previous Results" feature available in several REPLs of the survey (demonstrated by the variable `$1` in the above fragment). JShell and IPython ("Access to All") implement this feature as follows. Whenever a code snippet produces a result value (other than void), this result value is assigned to a fresh variable. For example, if the second snippet sent to IPython produces result value `5`, then the variable `_2` is assigned `5`. The behavior differs between JShell and IPython when a code snippet contains multiple statements. In IPython ("Last Output"), the result of a sequence of statements is the result of the last statement[5], e.g., the snippet **`print`**`(1);2;`**`print`**`(3)` prints `1` and `3` but has no result value. In JShell, the result of a sequence of statements is the result of each statement with a (non-void) result. If a snippet has multiple results, each result is assigned to a fresh variable. For example, if `3;2;System.out.println(1);` is sent as the first snippet to JShell, then the variables `$1` and `$2` are assigned the values `3` and `2` respectively and `1` is printed. In Node.js ("Access to Last"), a statement such as `console.log(1)` produces `undefined` as a result, which is then assigned to the variable _. PsySH also assigns the last uncaught exception to the variable `$_e`. This feature is helpful in situations where the exception is not easily reproduced, e.g., when caused by a (rare) non-deterministic, pseudorandom or timed event.

Most languages of the survey enable definitions to be redone ("Definition Modification"), with only iRB also allowing extensions to existing definitions ("Open & Extend"). The main challenge to redefining or modifying existing definitions is checking whether an updated definition is consistent with definitions that depend on it. This is particularly challenging for statically typed languages such as Java. In JShell, any inconsistencies are reported when a (now incorrect) definition is used, as shown by the following interaction:

```
jshell> class B {int mymethod(){return 0;}}
|  created class B
jshell> class A {int mymethod(){return new B().mymethod();}}
|  created class A
jshell> class B {long mymethod(){return 0;}}
|  replaced class B
jshell> int x = 4; int y = new A().mymethod(); int y = 5;
x ⇒ 4
|  attempted to use class A which cannot be instantiated or
|  its methods invoked until this error is corrected:
|   possible lossy conversion from long to int
|   class A { int mymethod() { return new B().mymethod(); }}
y ⇒ 5
```

[5]Even when void. A possible alternative is to use the last non-void result.

Note that the last snippet is neither type-checked and rejected as a whole nor that the error keeps the other statements from being executed. Statements appear to be type-checked individually, with any errors causing only the individual statement to be rejected. However, the following JShell interaction shows that this is a simplification:

```
jshell> int x = 1; new A(); int y = 2;
x ⇒ 1
|  Error:
|  cannot find symbol
|    symbol:  class A
```

A downside of showing inconsistencies just before they cause problems is that a menial mistake can cause a cascade of avoidable mistakes to go undetected, perhaps requiring tedious efforts to resolve. A downside of reporting inconsistencies as soon as they arrive is that they may be considered redundant and a nuisance when a programmer is aware and about to resolve the inconsistencies.

The C# REPL does not update method definitions affected by an update to another class. So when, in the example above, `mymethod` is called on a new instance of `A`, the behavior is that of the old `mymethod` of class `B`. (A similar example using fields rather than methods causes the C# REPL to hang.)

A general theme in the discussed language extensions is that they relate to the effects of code snippets on their successors. A REPL engineer should consider all the different kinds of (side-)effects code snippets can produce and decide for each effect whether it should propagate and, if so, how the programmer is informed of the effect, enabling them to update their mental model of the REPL's state. To help the programmer further, the ability to request an overview of the currently active bindings is desirable, especially together with a mechanism for inspecting (modified) type definitions.

6 Case Studies

This section discusses several REPL implementations for a number of languages with different user interfaces. The section is structured according to three case studies for the Rascal-defined languages MiniJava and QL, and the Haskell-defined language eFLINT. The case studies implement novel sequential variants of these languages.

6.1 A Jupyter Notebook for MiniJava

The MiniJava language is a subset of Java that retains the essential object-oriented features of Java [1, 6]. The semantics of a MiniJava program is given by its interpretation as a Java program. It is implemented as a definitional interpreter in the Rascal language workbench [15]. The extension to a sequential MiniJava uses Rascal's modular extension mechanisms and demonstrates the methodology of the previous section.

The first part of the extension is choosing the top-level constructs of the language. As for **JShell**, these are expressions, statements, variable, class, and method declarations, and their associative composition. The syntax of MiniJava is extended by adding the `Phrase` construct:

```
syntax Phrase
  = Expression ";" | Statement
  | VarDecl | ClassDecl | MethodDecl
  | assoc Phrase Phrase;
syntax Statement
  = ...
  | "throw" "new" StringLiteral ";";
syntax Expression
  = ...
  | Identifier "(" ExpressionList? ")";
```

The extension also includes a new method call variant, enabling (global) methods to be called without a receiver. The **`throw`**-keyword is added to demonstrate an implementation of handling uncaught exceptions. Exception values are simplified to string literals rather than than arbitrary objects.

The definitional interpreter of extended MiniJava is defined in Rascal as the function `Config eval(Phrase, Config)`, shown[6] in Figure 2. The type `Config`, shared by both MiniJava interpreters, is defined as the following tuple type:

```
alias Config = tuple[
  Env env, Sto sto,
  int seed, Out out,
  Val given, MaybeFailure failed,
  Val result
];
data MaybeFailure
  = failure(FailureType e)
  | no_failure()
  ;
data FailureType
  = failed()
  | exception(str msg)
  ;
```

Configurations have the following fields: the current execution environment (`env`), the store (`sto`), a seed (`seed`), the output of all executed phrases represented as a list of strings[7] (`out`), a given value (`given`) of type `Val` used for passing arguments, the field `failed` to indicate if and why the execution got 'stuck', and a value with the execution's result (`result`). The `Val` ADT (not shown) defines constructors for references, integers, booleans, vectors (arrays), environments, lists, closures, classes, objects, and `null`. The alternative `failed()` of `FailureType` indicates the execution got stuck because the

[6] The notation `(NT)\`...\`` is used to pattern match against or construct concrete syntax trees of type `NT`, where `NT` is some nonterminal defined in Rascal's native grammar formalism; the parts between fish-angle brackets represent typed holes of the pattern.

[7] The implementation converts the integers printed by MiniJava to strings and inserts a newline, corresponding to Java semantics.

```
Config eval((Phrase)`<Expression e> ;`, Config c)
  = catchExceptions(collectBindings(
      setOutput(createBinding(eval(c, e)))));

Config eval((Phrase)`<Statement s>`, Config c)
  = catchExceptions(collectBindings(
      setOutput(exec(s, c))));

Config eval((Phrase)`<ClassDecl cd>`, Config c)
  = catchExceptions(collectBindings(
      declareClass(cd, c)));

Config eval((Phrase)`<VarDecl vd>`, Config c)
  = catchExceptions(collectBindings(
      declareVariables(vd, c)));

Config eval((Phrase)`<MethodDecl md>`, Config c)
  = catchExceptions(collectBindings(
      declareGlobalMethod(md, c)));

Config eval((Phrase)`<Phrase p1> <Phrase p2>`, Config c)
  = eval(p2, eval(p1, c));
```

Figure 2. Interpreting MiniJava phrases.

evaluated program is invalid (e.g. due to unbound variables). The alternative `exception(str msg)` indicates an exception has been thrown with exception value `msg`.

The cases of Figure 2 that handle declarations (class, variable, or method) first produce an environment by calling the respective functions `declareClass`, `declareVariables` and `declareGlobalMethod`. These functions also produce output that informs the programmer of the successful binding of the respective class, variable or method. If a class is redefined, the programmer is also informed. The `collectBindings` function (not shown) adds the bindings in the computed environment (`result`) to the execution environment (`env`). The function `catchExceptions` (not shown) checks whether a phrase has failed or raised an exception. If so, the failure or exception is reported and removed, ensuring that the next phrase executes normally. Note that a MiniJava code snippet of the form `1;(2/0);3;` is parsed as a sequence of three phrases and not a code block consisting of three statements. Since the division by zero error is removed, the next phrase (`3;`) is executed normally. So, contrary to JShell, there is no distinction between phrases executed as separate code snippets or as a single, semi-colon separated code snippet. This arguably makes the language more consistent. The behavior of statements separated by a semi-colon in code blocks is unaffected and an exception will terminate the execution of a code block when it arises.

The first two cases of Figure 2 deal with expression and statement phrases, reusing the original interpreters for expressions and statements (`eval` and `exec` respectively). A statement, which may be a code block consisting of multiple statements, either computes `null` or an environment that contains the bindings for all variables that have been assigned a (new) value. The function `setOutput` (not shown) inspects the computed bindings, if any, and prints the variable and its assigned value, matching the behavior of JShell. An expression computes a value such as an integer, a boolean or an object reference. The function `createBinding` (not shown) assigns the computed value to a fresh variable, using the `seed` field of the current configuration, and binds the fresh variable to the identifier `$<i>`, where `<i>` is generated from the seed. The applications of `setOutput` and `collectBindings` ensure that the new binding is reported to the programmer and is active when the next phrase is executed, matching the behavior of JShell.

The final case confirms that two consecutive phrases are evaluated by function-composition. The implementation of method calls without receiver expression is not given.

The definitional interpreter of the extended language forms the interface to language services such as REPLs and computational notebooks. The connection between the definitional interpreter and Rascal's notebook framework Bacatá is discussed next.

Exploring Interpreters in Bacatá. Bacatá [22] is a generic Jupyter[8] kernel generator for languages developed within the Rascal Language Workbench. Bacatá is extended to support notebooks based on exploring interpreters. The generic implementation of the exploring interpreter maintains a full execution graph (in accordance to Definition 4.1). Bacatá relies on the definition of a language `repl`, a value of the `REPL` ADT shown below:

```
data REPL[&T]
= repl(&T initConfig, &T (str, &T) handler,
   Completion (str, int, &T) completor,
   Content (&T, &T) printer);
```

A value of `REPL` contains all required information to build a REPL command-line interface for a language, or, together with Bacatá, a computational notebook. The type parameter `&T` represents the configuration (e.g., `Config` of MiniJava). The `handler` takes a line of input and a configuration and produces a new configuration. The completor can be provided for tab-completion services. Finally, the `printer` produces (HTML) content from the previous and/or current configuration.

Bacatá is used as an interface between a Jupyter server and the language's REPL. The workflow that describes the communication among these components is as follows: Jupyter takes the user's code snippets and sends them to the language's interpreter through Bacatá. Bacatá takes the user's code and calls the language's `handler` (defined in the `repl` value), which is responsible for calling the parser and then the interpreter of the language. Finally, the `handler` produces a result, which is then displayed to the user, using the `printer`.

[8] http://jupyter.org/

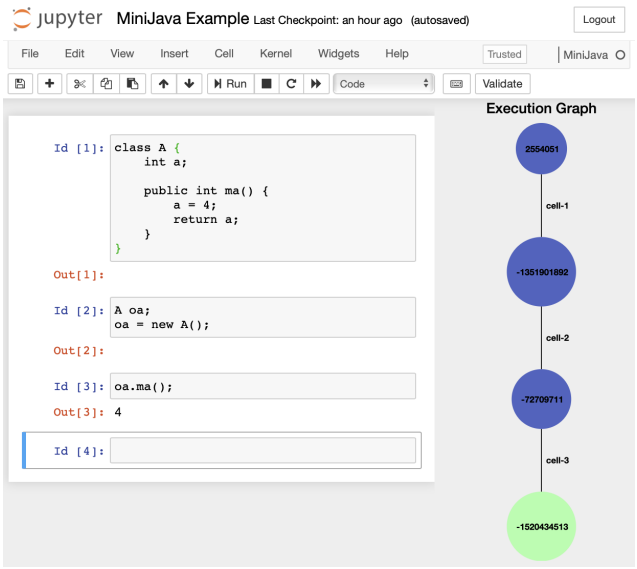

Figure 3. Notebook example.

Figure 3 shows a simple notebook for MiniJava, produced with Bacatá. The right shows the execution graph for exploring the user's interaction with the notebook. The active node is colored green. The user can click any other node, to make it active. The next cell will then be executed in the context of that very configuration, resulting in a split in the graph if the resulting configuration differs from the activated one.

A Notebook Interface for MiniJava. Obtaining a REPL-style command-line or notebook interface for MiniJava amounts to instantiating the `REPL` data type with the appropriate handlers, printers, and completors. In the case of a Bacatá-generated notebook Jupyter interface, the programmer has access to a visual representation of the execution graph of the exploring interpreter, as shown in Figure 3.

The handler for MiniJava parses the incoming input as a `Phrase` and calls the extended definitional interpreter, which returns a new configuration. The printer takes the old and new configuration and prints relevant output. After a successful execution, the differences between the `out` components of the new and old configuration is shown. In the case of a declaration, the difference between the two `env` components gives the new bindings. The completor uses the bindings in its input configuration to suggest possible completions for identifiers.

6.2 QL: A DSL for Questionnaires

QL is a little language for defining interactive questionnaires [8, 9], like tax filing forms or online surveys. A QL form defines a sequence of questions, where each question has a label, an identifier, a type (boolean, integer, or string), and an optional expression if the question is computed. Expressions contain the usual arithmetic and comparison operations, and

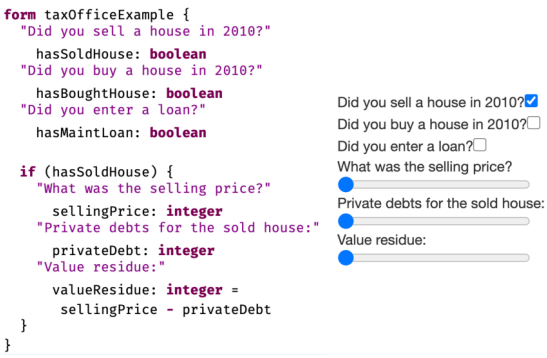

Figure 4. A QL questionnaire (left) and its rendering (right).

allow referring to the current value of another question. Furthermore, questions can be made conditional using if-then and if-then-else constructs.

The meaning of a QL program is a rendering as an interactive GUI program, where the user enters values for the (non-computed) questions. Depending on this input, conditional questions may be shown or hidden, and the value of computed questions may be recomputed, similar to a spreadsheet. A simple example is shown in Figure 4, including its rendering as an interactive UI.

From a REPL perspective, QL is interesting, because a form specifies a conditional data-flow network rather than a program consisting of instructions. Nevertheless, in this section we introduce a prototype REPL for QL, both as an instructive thought experiment, and to stress the concept of sequential language.

Abstractly, the semantics of QL can be described with the following (Rascal) function signature:

```
tuple[UI, Env] eval(Form form, Env env, Event evt);
```

Given a form, an environment mapping question identifiers to values (`Env`), and a user event (`Event`), the function `eval` produces a rendering (`UI`) and an updated environment. Running a QL questionnaire then amounts to constructing an initial rendering, and then updating the current environment and redrawing the UI after every user action.

To provide a REPL interface for QL, we extend the language with a new start nonterminal, `Cmd`, the definition of which is shown in Figure 5. Commands are the snippets that the user can enter at the command line.

The first four alternatives of `Cmd` capture constructs to manipulate forms. The user can define complete forms, append or prepend individual questions to the current form, and replace questions arbitrarily nested in the form using a positional reference mechanism (`Addr`).

The last two alternatives can be used to evaluate expressions, which shows the result, or update the value of a (non-computed) question, if the current state of the UI allows it. The update-value action simulates a user interaction if

```
syntax Cmd
  = Form                // define form
  | Question            // append a question
  | Question "..."      // prepend question
  | "@" Addr Question // replace question
  | Expr                // evaluate expression
  | Id "=" Value;       // perform user action

syntax Script
  = Cmd* commands       // batch perform commands
```

Figure 5. Language extension for ReplizedQL.

the form would have been rendered as a proper UI. Finally, Figure 5 defines a `Script` nonterminal to combine multiple commands in sequence.

The interpreter for commands is a function from a command and the current configuration to a new configuration:

```
Config eval(Cmd cmd, Config cfg) { ... }
```

The `Config` type captures the current environment, the current form, and a list of output values (UI renderings and expression evaluation results).

That our definition of QL is sequential can be seen from the definition of the interpreter for `Script`s:

```
Config eval(Script scr, Config cfg) =
  ( cfg | eval(cmd, it) | Cmd cmd ← scr.commands );
```

This function simply composes the `eval` function for commands for every command in the script[9]. This follows the definition of sequential language of Section 3.

A Sample Interaction. The above interpreter for commands can be hooked to Rascal's standard REPL infrastructure to obtain a command-line interface for QL. We illustrate the semantics of sequential QL below, using a sample user interaction. The code snippets use `>` as prompt, the output of a command is shown directly below.

First, let's define a simple form:

```
> form simple { }
  ·
```

The result is the empty rendering of the UI, indicated by ·. Then we append a (computed question), labeled "A", of type integer:

```
> "A" a: integer = c + b + 1
  A ·
```

The `a` question is not conditional, so it is shown in the UI rendering; note however that the value of the question is still undefined because questions `c` and `b` have not yet been defined.

[9]The notation `( init | ... it ... | gen )` is Rascal syntax for writing a reduce operation.

The `b` question could be defined as follows:

```
> if (a < 20) "B" b: integer = c + 1
  A ·
```

Since `b` is still undefined (because `c` is), `a` remains undefined as well, and as a result, the visibility condition of `b` evaluates to false. This all changes, however, after defining `c`:

```
> if (a > 20) "C" c: integer
  A 2
  B 1
```

The question `c` is not computed, so it receives an initial default value (in this case 0). Both `a` and `b` can now be computed, as well as the condition of `b`, causing `b` to be shown in the UI. Now let's change the value of `c`:

```
> c = 10
  A 22
  C [10]
```

Setting `c` to 10 disables `b`, but changes the visibility condition of `c` to true, making it appear in the UI. The square brackets around the value of `c` indicate it is editable.

Changing the value of `c` to 5 updates the UI accordingly:

```
> c = 5
  A 12
  B 6
```

Now `b` becomes visible, and `c` is hidden again.

It is possible to add questions to the beginning of the form:

```
> "D" d: integer = 3 * a...
  D 36
  A 12
  B 6
```

Or using the path-based address notation:

```
> :form
form simple {
 [0] "D" d: integer = 3 * a
 [1] "A" a: integer = c + b + 1
 [2] if (a < 20)
  [2.0] "B" b: integer = c + 1
 [3] if (a > 20)
  [3.0] "C" c: integer
}
> @2.0 "c + 1 is:" b: integer = c + 1
  D 36
  A 12
  c + 1 is: 6
```

The `:form` meta-command pretty-prints the current form annotated with addresses for every question. Using the `@`-notation, the user can replace any question in the form, in this case to change the label of the `b` question.

Note that the append-, prepend-, and position-based adding and replacement of questions can be considered a rather low-level (maybe even pathological) way of editing a program

```
#0 > Fact person. Placeholder parent,child For person
new fact-type person
no enabled actions or events
#3 > +person(Alice). +person(Bob) // introduce persons
+"Alice":person
+"Bob":person
no enabled actions or events
#5 > Fact parent-of Identified by parent * child
new fact-type parent-of
no enabled actions or events
#6 > +parent-of(Alice,Bob)
+("Alice":person,"Bob":person):parent-of
no enabled actions or events
#7 > Act call-for-help Actor child Recipient parent
                       Holds when parent-of()
new fact-type call-for-help
+("Bob":person,"Alice":person):call-for-help
enabled actions & events:
1. ("Bob":person,"Alice":person):call-for-help
#8 > :choose 1     // Bob asks Alice for help
enabled actions & events:
1. ("Bob":person,"Alice":person):call-for-help
#9 > :revert 7     // to before the action was declared
+("Alice":person,"Bob":person):parent-of
#7 > :current      // show the current set of facts
"Alice":person
"Bob":person
("Alice":person,"Bob":person):parent-of
#7 > ?Enabled(call-for-help(Bob,Alice)) // query
undeclared type: call-for-help
```

Figure 6. A session with the eFLINT command-line REPL.

(reminiscent of the line-based editors of the past). Nevertheless, without necessesarily claiming this is a realistic way of evolving programs, it does illustrate a kind of REPL "completeness", where every program and program change can be realized using commands at the prompt.

6.3 eFLINT: Executable Normative Specifications

eFLINT is a DSL for developing executable normative specifications used to reason about compliance with regulations, contracts and/or policies [43]. eFLINT programs are used to simulate or verify normative decision making processes. The methodology of Section 5 has been applied to develop two REPLs on top of one exploring interpreter for eFLINT. The implementation of eFLINT is available at GitLab [42].

REPL Interfaces. The first REPL is a command-line tool for exploring compliant and non-compliant behavior. Figure 6 shows an example session where the user explores the norm "children can ask their parents for help". As a meta-command, a user can choose actions and events to trigger from a given list of options. Choosing an action or event has the effect of updating a database of 'facts', representing the state of the world at a particular moment in time. A fact is said to 'hold true' if it is present in the database. Some facts are reifications of actions and correspond to acceptable behavior when they hold true and when they are enabled by their pre-conditions. Disabled actions can be executed in order to explore non-compliant behavior (although causing a violation). Other facts represent duties, which need to be 'terminated' before one of their violation conditions holds true. The phrases of the language are declarations of fact-, act-, event- and duty-types, action or event triggers, insertion and removal of facts and queries on the database. After a phrase is executed, the user is presented with the changes in the database, newly defined types, any violations and a new list of options.

The second REPL is a TCP server that listens on a chosen port for incoming phrases and responds with the same information as the command-line REPL (in JSON form). The TCP server is used as a general method for connecting other languages with eFLINT to benefit from the normative specification written in eFLINT. For example, a program can send queries to the eFLINT server to check whether certain actions are enabled before actually performing them. In this way, software can be developed that is 'compliant by design'.

The REPLs are developed on top of an exploring interpreter for eFLINT, briefly explained next.

Execution Tree. The type *Explorer* is an alias for functions that receive an *Instruction* and return a *Response* in the *IO* monad (Haskell's mechanism for input and output).

```
type Explorer    = Instruction → IO Response
data Instruction = Execute CPhrase | Revert Int | Display
data Response    = Success Node CPhrase Node | ExecError Error
type Node        = (Int, Config)
```

The values of *Instruction* correspond to the actions of the generic exploring interpreter algorithm. There are two types of response, for successful executions and failing executions respectively. One of the values of *Error* indicates that the integer given as part of some **revert** action does not correspond to a known configuration. The success response contains the elements of an edge in the execution graph: two nodes and a label (phrase). The edge gives the effects, in terms of an input and output configuration, of the last phrase executed by the exploring interpreter. A node is a configuration and an integer that uniquely identifies the node. The label is a value of type *CPhrase*, a phrase that has been compiled.

A configuration contains information about declared types (a type environment), a database of facts and a list of output holding any reported violations:

```
data Config = Cfg { tyenv :: TyEnv, state :: Set Fact, out :: [String] }
```

The algorithm maintains a tree rather than a graph, and does so in a way that makes it very simple to find the path from the root to any given configuration in the tree. The type

SIDMap is an alias for a map mapping integers to the configurations with which they form a node. The type *History* represents a tree as a collection of edges.

type *SIDMap* = *IntMap Config*
type *History* = *IntMap* (*Int*, *CPhrase*)

If x maps to (y, p) in the *History* map, this means that there is an edge $\langle \gamma, p, \gamma' \rangle$ in the tree where y is the integer identifying γ and x is the integer identifying γ'.

REPL Features. The function *getPath*::*Int* → *SIDMap* → *History* → [*CPhrase*] receives an integer identifying a node and uses the maps to compute the sequence of phrases labeling the path from the root of the tree to the node. The function is used to save a session by pretty-printing and storing the returned phrases in a file.

The definitional interpreter of eFLINT receives compiled phrases (*CPhrase*s) as input. The tool-set for eFLINT contains a compiler that translates from *Phrase* to *CPhrase*. The compiler checks whether a *Phrase* is well-typed and applies conversions to make explicit certain implicit operator applications. Compilation is performed by the function *compile* : *TypeEnv* → *Phrase* → *CPhrase*, receiving as input the type environment of the current configuration held by the exploring interpreter.

When the command-line or TCP server REPL receives a *String* for execution, the string is parsed as a *Phrase*. If successful, the *Phrase* is type-checked and compiled to a *CPhrase*. The *CPhrase* is sent as an **execute** action to the exploring interpreter, which invokes the definitional interpreter and responds either with an error or with the edge of its graph representing the latest execution. This edge is given to a function called *effectsOf* to compute the effects of executing the phrase. The function *effectsOf* finds any new bindings by computing the difference between the two type environments of the input configurations, finds any created or terminated facts by computing the difference between the two state components and finds new violations by computing the difference between the two output components.

7 Discussion & Related Work

Limitations & Future Work. The techniques described in this paper are applicable to languages that can be implemented by deterministic interpreters with explicit state representations Moreover, if an execution graph is not needed, then state does not have to be represented explicitly (see Section 5.1), as long as the effects of top-level phrases still compose and are communicated clearly to REPL users. This requirement does not necessarily rule out concurrent, non-deterministic, compiled or data flow languages. In some cases it is possible to *model* the complicating aspects of these languages, e.g., with thread models, data flow graphs and lists to capture non-deterministic results.

Purely functional interpreters with explicit state representation are, however, further removed from actual implementations and may be less suitable for developing practical REPLs. For instance, a definitional interpreter for C can model memory (pointers) rather than providing real memory access. A REPL for C can also be based on an interpreter that invokes a C compiler, wrapping current and previous code snippets in `int main() {...}`, before compiling and executing the resulting program (similar to the Go REPL). It is possible to obtain a REPL interface in this way, but it would not be based on a sequential language and the explorative quality of exploring interpreters is lost. The applicability of our approach in the context of such compilation-based REPLs is to be investigated further.

The interpreters discussed in this paper are all implemented in functional programming languages (Rascal and Haskell) with immutable data. Maintaining the execution graph is therefore easy to implement, but it may come at a cost of performance and memory footprint. Further research is needed to represent the graph more efficiently, for instance by maximizing sharing, caching intermediate results, or selectively culling the graph. The pragmatics of a REPL (small snippets, immediate feedback, etc.), however, suggest that such optimization might be premature.

Although not shown in this paper, exploring interpreters can also be used to realize additional features not typically found in REPLs by performing sequences of **execute** and **revert** actions in response to a single user action. For example, if a user edits a cell in a notebook, this could cause the exploring interpreter to revert to the configuration in which that cell was originally executed, keeping track of all cells undone this way, re-executing the (now modified) cell, and executing all the remembered cells in the order they were first executed. Further research is needed to establish how this relates to live programming [39, 44]. The QL language described in Section 6.2 has a live programming environment and forms a natural staring point for this study.

The MiniJava notebook discussed in Section 6.1 displays the execution graph of the exploring interpreter, allowing arbitrary roll-backs to explore alternative execution paths. In future work we will explore the ability of the exploring interpreter to support exploratory programming. More generally, we aim to describe algebraic operations over execution graphs for both live and exploratory programming.

The methodology of Section 5 starts from a single base language. The methodology is easily generalized to take multiple based languages as a starting point and defining a single sequential language as an extension of all the base languages, which is then used as the basis for a so-called *polyglot* REPL. The definitional interpreter for the sequential extension may not be easy to define, however, when the effects of the phrases of the different base languages are not easily reconciled. In a future study we hope to formulate and

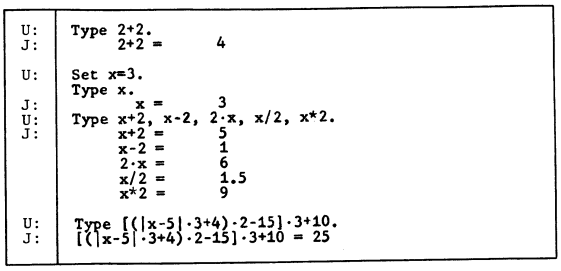

Figure 7. Early user interaction using JOSS [37]

demonstrate the more general methodology and to show its benefits to developing polyglot REPLs and notebooks.

Related Work. REPLs have long history and documentation on this history is scattered across sources. The Flexowriter system of Lisp I from 1960 is perhaps the oldest REPL implementation [21]. An early description of REPL behavior can be found in Peter Deutsch's memo on PDP-1 LISP [7]:

> *Each S-expression typed in will be evaluated and its value printed out.*

The PILOT system [40] is one of the earliest and most advanced interactive REPL systems, also based on a LISP, in that it supports fully incremental and interactive evolution of programs. Teitelman writes that REPL-style interaction with Interlisp happened with the introduction of time-sharing at MIT in 1964 [41]. It is very well possible, however, that earlier Lisps and pre-1968 FORTH implementations [32] had REPL interfaces as well. The earliest programming language REPL that is not a Lisp we could find documentation of is the JOHNNIAC Open-Shop System (JOSS) [37]. Figure 7 shows an example of interacting with JOSS.

REPLs have a close relation to computational notebooks, which were pioneered in the Mathematica system [47]. More recently, this style has been adopted in the context of other programming languages. IPython [28] and Jupyter [16] provide a means for computational story telling, where cells containing code are interleaved with output and prose cells. The language workbench framework Bacatá allows a language engineer to provide a notebook feature by reusing existing language artifacts [22]. In Section 6.1 we have adapted Bacatá to include the generic exploring interpreter algorithm of which the execution graph is shown in the notebook

Reynolds first employed definitional interpreters as a vehicle for reasoning about languages [33, 34]. His analysis took advantage of the formal similarity between denotational and interpretative semantics [35]. The formal similarity between various approaches to formal semantics is captured by Initial Algebra Semantics [10]. Modular extension mechanisms have been developed for semantics, such as monad transformers [18, 25], entity propagation in Modular Structural Operational Semantics [3], and copy-rules and forwarding in Attribute Grammars[38, 45]. These mechanisms greatly enhance the practicality of definitional interpreters. In modern languages, we see advanced use of monads in Haskell [20, 29], Object Algebras [27] in Java, C# and Scala and intrinsically-typed definitional interpreters in Agda [36].

The usage of an execution graph containing all intermediate configurations of a user's interaction is related to back-in-time debugging [19, 31], also known as omniscient debugging [5, 17], allowing programmers to go back in time of an execution history. In contrast, however, exploring interpreters allow users to go back in *session time*, obtaining both a new run-time state and program state.

Conclusion. REPLs provide programmers with a direct interface to a programming language, supporting exploration, testing, and incremental development. All mainstream languages have REPL interfaces, indicating the value they represent to programmers. However, the actual language that is accepted by the REPL is often not well-defined, and engineering REPLs lacks solid design principles.

In this paper we have surveyed existing REPLs in a feature-oriented domain analysis, showing a wide diversity in feature support. To make the relation between a REPL and its language precise, we have defined and formalized the notion of sequential language, and used it as the basis of a methodology to construct REPL interpreters. The versatility of the approach has been demonstrated in three case studies, one based on MiniJava, and two based on DSLs (QL and eFLINT). The case studies show notebook, command-line, and client-server REPL interfaces, developed using the methodology by extending base languages and reusing existing interpreters.

The concept of sequential language and its associated language design and engineering guidelines may provide better insight into the essence of REPLs, and promote a principled approach to the construction of REPLs.

Acknowledgements. We would like to thank the Twitter hive mind, and Rainer Joswig in particular, for help in navigating the early history of REPLs and the anonymous reviewers for their helpful comments.

This work is supported by the NWO project (628.009.014) Secure Scalable Policy-enforced Distributed Data Processing (SSPDDP), part of the NWO research program Big Data: Real Time ICT for Logistics

This work was partially executed in the context of the CWI/INRIA Associate Team Agile Language Engineering (ALE).

References

[1] Andrew W. Appel and Jens Palsberg. 2003. *Modern Compiler Implementation in Java* (2nd ed.). Cambridge University Press.

[2] Egidio Astesiano. 1991. Inductive and Operational Semantics. In *IFIP State-of-the-Art Reports, Formal Descriptions of Programming Concepts*, E.J. Neuhold and M. Paul (Eds.). Springer, 51–136.

[3] Casper Bach Poulsen and Peter D. Mosses. 2014. Generating Specialized Interpreters for Modular Structural Operational Semantics. In

23rd International Symposium on Logic-Based Program Synthesis and Transformation. Springer, 220–236. https://doi.org/10.1007/978-3-319-14125-1_13

[4] Bas Basten, Jeroen van den Bos, Mark Hills, Paul Klint, Arnold Lankamp, Bert Lisser, Atze van der Ploeg, Tijs van der Storm, and Jurgen Vinju. 2015. Modular language implementation in Rascal – experience report. *Science of Computer Programming* 114 (2015), 7–19. https://doi.org/10.1016/j.scico.2015.11.003

[5] Erwan Bousse, Dorian Leroy, Benoît Combemale, Manuel Wimmer, and Benoit Baudry. 2018. Omniscient debugging for executable DSLs. *Journal of Systems and Software* 137 (2018), 261–288. https://doi.org/10.1016/j.jss.2017.11.025

[6] João Cangussu, Jens Palsberg, and Vidyut Samanta. 2002. The MiniJava Project. https://www.cambridge.org/us/features/052182060X. [Online, accessed 12 October 2020].

[7] Peter Deutsch. 1964. *PDP-1 LISP*. Technical Report. MIT Research Laboratory for Electronics. http://www.bitsavers.org/pdf/mit/rle_pdp1/memos/Deutsch_PDP-1_LISP.pdf.

[8] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. 2013. The State of the Art in Language Workbenches - Conclusions from the Language Workbench Challenge. In *Proceedings of the 6th International Conference on Software Language Engineering (SLE'13)*. 197–217. https://doi.org/10.1007/978-3-319-02654-1_11

[9] Sebastian Erdweg, Tijs van der Storm, Markus Volter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriel Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. 2015. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures* 44 (2015), 24 – 47. https://doi.org/10.1016/j.cl.2015.08.007

[10] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. 1977. Initial Algebra Semantics and Continuous Algebras. *J. ACM* 24, 1 (Jan. 1977), 68–95. https://doi.org/10.1145/321992.321997

[11] Maria Gouseti, Chiel Peters, and Tijs van der Storm. 2014. Extensible Language Implementation with Object Algebras (Short Paper). In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences (GPCE'14)* (Västerås, Sweden). 25–28. https://doi.org/10.1145/2658761.2658765

[12] Pierre Jeanjean, Benoit Combemale, and Olivier Barais. 2019. From DSL Specification to Interactive Computer Programming Environment. In *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering (SLE'19)* (Athens, Greece). 167–178. https://doi.org/10.1145/3357766.3359540

[13] Gilles Kahn. 1987. Natural Semantics. In *Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science*. Springer-Verlag, 22–39. https://doi.org/10.1007/BFb0039592

[14] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. 1990. *Feature-oriented domain analysis (FODA) feasibility study*. Technical Report. Carnegie-Mellon Software Engineering Institute.

[15] Paul Klint, Tijs van der Storm, and Jurgen Vinju. 2009. Rascal: A Domain Specific Language for Source Code Analysis and Manipulation. In *Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE Computer Society, 168–177. https://doi.org/10.1109/SCAM.2009.28

[16] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. 2016. Jupyter Notebooks – a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*. 87–90.

[17] Bil Lewis. 2003. Debugging Backwards in Time. *Computing Research Repository* cs.SE/0310016 (2003). http://arxiv.org/abs/cs/0310016

[18] Sheng Liang, Paul Hudak, and Mark Jones. 1995. Monad Transformers and Modular Interpreters. In *22nd Symposium on Principles of Programming Languages*. ACM, 333–343. https://doi.org/10.1145/199448.199528

[19] Adrian Lienhard, Tudor Gîrba, and Oscar Nierstrasz. 2008. Practical object-oriented back-in-time debugging. In *European Conference on Object-Oriented Programming*. Springer, 592–615.

[20] Simon Marlow. 2010. Haskell 2010 Language Report. https://www.haskell.org/onlinereport/haskell2010/. [Online, accessed 12 October 2020].

[21] J. McCarthy, R. Brayton, D. Edwards, P. Fox, L. Hodes, D. Luckham, K. Maling, and D. Park S. Russell. 1960. *Lisp I programmer's manual*. Computation Center and Research Laboratory of Electronics (MIT). http://history.siam.org/sup/Fox_1960_LISP.pdf. [Online, accessed 12 October 2020].

[22] Mauricio Verano Merino, Jurgen J. Vinju, and Tijs van der Storm. 2020. Bacatá: Notebooks for DSLs, Almost for Free. *Programming Journal* 4, 3 (2020), 11. https://doi.org/10.22152/programming-journal.org/2020/4/11

[23] Marjan Mernik, Jan Heering, and Anthony M. Sloane. 2005. When and How to Develop Domain-specific Languages. *Comput. Surveys* 37, 4 (2005), 316–344. https://doi.org/10.1145/1118890.1118892

[24] Robin Milner, Mads Tofte, and David MacQueen. 1997. *The Definition of Standard ML*. MIT Press.

[25] Eugenio Moggi. 1991. Notions of Computation and Monads. *Information and Computation* 93, 1 (1991), 55–92. https://doi.org/10.1016/0890-5401(91)90052-4

[26] Peter D. Mosses. 2004. Modular Structural Operational Semantics. *Journal of Logic and Algebraic Programming* 60–61 (2004), 195–228. https://doi.org/10.1016/j.jlap.2004.03.008

[27] Bruno C. d. S. Oliveira and William R. Cook. 2012. Extensibility for the Masses - Practical Extensibility with Object Algebras. In *ECOOP 2012 – Object-Oriented Programming*. Springer Berlin Heidelberg, 2–27.

[28] Fernando Perez and Brian E. Granger. 2007. IPython: A System for Interactive Scientific Computing. *Computing in Science and Engineering* 9, 3 (2007), 21–29. https://doi.org/10.1109/MCSE.2007.53

[29] Matthew Pickering, Nicolas Wu, and Csongor Kiss. 2019. Multi-stage programs in context. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2019, Berlin, Germany, August 18-23, 2019*, Richard A. Eisenberg (Ed.). ACM, 71–84. https://doi.org/10.1145/3331545.3342597

[30] Gordon D. Plotkin. 2004. A Structural Approach to Operational Semantics. *Journal of Logic and Algebraic Programming* 60–61 (2004), 17–139. https://doi.org/10.1016/j.jlap.2004.05.001 Reprint of Technical Report FN-19, DAIMI, Aarhus University, 1981.

[31] Guillaume Pothier, Éric Tanter, and José Piquer. 2007. Scalable omniscient debugging. *ACM SIGPLAN Notices* 42, 10 (2007), 535–552. https://doi.org/10.1145/1297105.1297067

[32] Elizabeth D. Rather, Donald R. Colburn, and Charles H. Moore. 1993. The Evolution of Forth. In *The Second ACM SIGPLAN Conference on History of Programming Languages* (Cambridge, Massachusetts, USA) *(HOPL-II)*. ACM, 177–199. https://doi.org/10.1145/154766.155369

[33] John C. Reynolds. 1972. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the ACM Annual Conference - Volume 2* (Boston, Massachusetts, USA). 717–740. https://doi.org/10.1145/800194.805852

[34] John C. Reynolds. 1998. Definitional Interpreters for Higher-Order Programming Languages. *Higher-Order and Symbolic Computation* 11, 4 (1998), 363–397. https://doi.org/10.1023/A:1010027404223

[35] John C. Reynolds. 1998. Definitional Interpreters Revisited. *Higher-Order and Symbolic Computation* 11, 4 (1998), 355–361. https://doi.org/10.1023/A:1010075320153

[36] Arjen Rouvoet, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. 2020. Intrinsically-Typed Definitional Interpreters for Linear, Session-Typed Languages. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP'20)*. 284–298. https://doi.org/10.1145/3372885.3373818

[37] J. C. Shaw. 1964. JOSS: A designer's view of an experimental on-line computing system. In *AFZPS Conference Proceedings, vol. 26, 1964 Fall Joint Computer Conference*. 455–464.

[38] S. Doaitse Swierstra, Pablo R. Azero Alcocer, and João Saraiva. 1999. Designing and implementing combinator languages. In *Advanced Functional Programming*. Springer Berlin Heidelberg, 150–206. https://doi.org/10.1007/10704973_4

[39] Steven L. Tanimoto. 2013. A perspective on the evolution of live programming. In *1st International Workshop on Live Programming (LIVE'13)*. IEEE, 31–34. https://doi.org/10.1109/LIVE.2013.6617346

[40] Warren Teitelman. 1966. *PILOT: A Step Toward Man-Computer Symbiosis*. Ph.D. Dissertation. MIT. http://hdl.handle.net/1721.1/6905

[41] Warren Teitelman. 2008. History of Interlisp. In *Celebrating the 50th Anniversary of Lisp* (Nashville, Tennessee). ACM, Article 5, 5 pages. https://doi.org/10.1145/1529966.1529971

[42] L. Thomas van Binsbergen. 2020. eFLINT implementation on GitLab. https://gitlab.com/calculemus-flint/eflint-tools/eflint. [Online, accessed 12 October 2020].

[43] L. Thomas van Binsbergen, Lu-Chi Liu, Robert van Doesburg, and Tom van Engers. 2020. eFLINT: A Domain-Specific Language for Executable Norm Specifications. In *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2020)*. ACM. https://doi.org/10.1145/3425898.3426958

[44] Tijs van der Storm. 2013. Semantic deltas for live DSL environments. In *1st International Workshop on Live Programming (LIVE'13)*. IEEE, 35–38. https://doi.org/10.1109/LIVE.2013.6617347

[45] Eric Van Wyk, Oege de Moor, Kevin Backhouse, and Paul Kwiatkowski. 2002. Forwarding in Attribute Grammars for Modular Language Design. In *Compiler Construction*, R.Nigel Horspool (Ed.). Lecture Notes in Computer Science, Vol. 2304. Springer Berlin Heidelberg, 128–142. https://doi.org/10.1007/3-540-45937-5_11

[46] Michał Walicki and Sigurd Meldal. 1997. Algebraic Approaches to Nondeterminism – an Overview. *Comput. Surveys* 29, 1 (March 1997), 30 – 81. https://doi.org/10.1145/248621.248623

[47] Stephen Wolfram. 1999. *The Mathematica Book (4th Edition)*. Cambridge University Press, USA.